Essential Shell Programming

Definition:

Shell is an agency that sits between the user and the UNIX system.

Description:

Shell is the one which understands all user directives and carries them out. It processes the commands issued by the user. The content is based on a type of shell called Bourne shell.

Shell Scripts

When groups of command have to be executed regularly, they should be stored in a file, and the file itself executed as a shell script or a shell program by the user. A shell program runs in interpretive mode. It is not complied with a separate executable file as with a C program but each statement is loaded into memory when it is to be executed. Hence shell scripts run slower than the programs written in high-level language. .sh is used as an extension for shell scripts. However the use of extension is not mandatory.

Shell scripts are executed in a separate child shell process which may or may not be same as the login shell.

Example: script.sh

```
#! /bin/sh
# script.sh: Sample Shell Script
echo “Welcome to Shell Programming”
echo “Today’s date : `date`”
echo “This months calendar:”
cal ‘date “+%m 20%y”‘                    #This month’s calendar.
echo “My Shell :$ SHELL”
```

The # character indicates the comments in the shell script and all the characters that follow the # symbol are ignored by the shell. However, this does not apply to the first line which beings with #. This because, it is an interpreter line which always begins with #! followed by the pathname of the shell to be used for running the script. In the above example the first line indicates that we are using a Bourne Shell.

To run the script we need to first make it executable. This is achieved by using the chmod command as shown below:

```
$ chmod +x script.sh
```

Then invoke the script name as:

$ script.sh

Once this is done, we can see the following output :

Welcome to Shell Programming

Today's date: Mon Oct 8 08:02:45 IST 2007

This month's calendar:

October 2007

Su	Mo	Tu	We	Th	Fr	Sa
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

My Shell: /bin/Sh

As stated above the child shell reads and executes each statement in interpretive mode. We can also explicitly spawn a child of your choice with the script name as argument:

sh script.sh

Note: Here the script neither requires a executable permission nor an interpreter line.

Read: Making scripts interactive

The read statement is the shell's internal tool for making scripts interactive (i.e. taking input from the user). It is used with one or more variables. Inputs supplied with the standard input are read into these variables. For instance, the use of statement like

read name

causes the script to pause at that point to take input from the keyboard. Whatever is entered by you will be stored in the variable *name*.

Example: A shell script that uses read to take a search string and filename from the terminal.

```
#! /bin/sh
# emp1.sh: Interactive version, uses read to accept two inputs
#
echo “Enter the pattern to be searched: \c”                    # No newline
read pname
echo “Enter the file to be used: \c”                           # use echo –e in bash
read fname
echo “Searching for pattern $pname from the file $fname”
grep $pname $fname
echo “Selected records shown above”
```

Running of the above script by specifying the inputs when the script pauses twice:

```
$ emp1.sh
Enter the pattern to be searched : director
Enter the file to be used: emp.lst
Searching for pattern director from the file emp.lst

9876    Jai Sharma      Director    Productions
2356    Rohit           Director    Sales
Selected records shown above
```

Using Command Line Arguments

Shell scripts also accept arguments from the command line. Therefore e they can be run non interactively and be used with redirection and pipelines. The arguments are assigned to special shell variables. Represented by $1, $2, etc; similar to C command arguments argv[0], argv[1], etc. The following table lists the different shell parameters.

Shell parameter	Significance
$1, $2…	Positional parameters representing command line arguments
$ #	No. of arguments specified in command line
$ 0	Name of the executed command
$ *	Complete set of positional parameters as a single string
"$ @"	Each quoted string treated as separate argument
$?	Exit status of last command
$$	Pid of the current shell
$!	PID of the last background job.

Table: shell parameters

exit and Exit Status of Command

To terminate a program exit is used. Nonzero value indicates an error condition.

Example 1:

$ cat foo

Cat: can't open foo

Returns nonzero exit status. The shell variable $? Stores this status.

Example 2:

grep director emp.lst > /dev/null:echo $?
0

Exit status is used to devise program logic that braches into different paths depending on success or failure of a command

The logical Operators && and ||

The shell provides two operators that aloe conditional execution, the && and ||.

Usage:

cmd1 && cmd2

cmd1 || cmd2

&& delimits two commands. cmd 2 executed only when cmd1 succeeds.

Example1:

$ grep ‘director’ emp.lst && echo “Pattern found”

Output:

9876 Jai Sharma Director Productions

2356 Rohit Director Sales

Pattern found

Example 2:

$ grep ‘clerk’ emp.lst || echo “Pattern not found”

Output:

Pattern not found

Example 3:

grep “$1” $2 || exit 2

echo “Pattern Found Job Over”

The if Conditional

The if statement makes two way decisions based on the result of a condition. The following forms of if are available in the shell:

Form 1

if *command is successful*
then
execute commands
fi

Form 2

if *command is successful*
then
execute commands
else
execute commands
fi

Form 3

if *command is successful*
then
execute commands
elif *command is successful*
then...
else...
fi

If the command succeeds, the statements within if are executed or else statements in else block are executed (if else present).

Example:

```
#! /bin/sh
if grep "^$1" /etc/passwd 2>/dev/null
then
        echo "Pattern Found"
else
        echo "Pattern Not Found"
fi
```

Output1:

```
$ emp3.sh ftp
ftp: *.325:15:FTP User:/Users1/home/ftp:/bin/true
Pattern Found
```

Output2:

$ emp3.sh mail

Pattern Not Found

While: Looping

To carry out a set of instruction repeatedly shell offers three features namely while, until and for.

Syntax:

```
while condition is true
do
        Commands
done
```

The commands enclosed by do and done are executed repeatedly as long as condition is true.

Example:

```
  #! /bin/usr
 ans=y
while [“$ans”=”y”]
do
        echo “Enter the code and description : \c” > /dev/tty
        read code description
        echo “$code $description” >>newlist
        echo “Enter any more [Y/N]”
        read any
        case $any in
        Y* | y* ) answer =y;;
        N* | n*) answer = n;;
                *) answer=y;;
        esac
done
```

Input:

Enter the code and description : 03 analgestics

Enter any more [Y/N] :y

Enter the code and description : 04 antibiotics

Enter any more [Y/N] : [Enter]

Enter the code and description : 05 OTC drugs

Enter any more [Y/N] : n

Output:

$ cat newlist

03 | analgestics

04 | antibiotics

05 | OTC drugs

Using test and [ ] to Evaluate Expressions

Test statement is used to handle the true or false value returned by expressions, and it is not possible with if statement. Test uses certain operators to evaluate the condition on its right and returns either a true or false exit status, which is then used by if for making decisions. Test works in three ways:

- Compare two numbers
- Compares two strings or a single one for a null value
- Checks files attributes

Test doesn't display any output but simply returns a value that sets the parameters $?

Numeric Comparison

Operator	Meaning
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-ge	Greater than or equal to
-lt	Less than
-le	Less than or equal

Table: Operators

Operators always begin with a – (Hyphen) followed by a two word character word and enclosed on either side by whitespace.

Numeric comparison in the shell is confined to integer values only, decimal values are simply truncated.

Ex:
$x=5;y=7;z=7.2

1. $test $x –eq $y; echo $?
   1 *Not equal*

2. $test $x –lt $y; echo $?
   0 *True*

3. $test $z –gt $y; echo $?
   1 *7.2 is not greater than 7*
   2
4. $test $z –eq $y ; echo $y
   0 *7.2 is equal to 7*
   1

Example 3 and 4 shows that test uses only integer comparison.

The script emp.sh uses test in an if-elif-else-fi construct (Form 3) to evaluate the shell parameter $#

```
#!/bin/sh
#emp.sh: using test, $0 and $# in an if-elif-else-fi construct
#
If test $# -eq 0; then
Echo “Usage : $0 pattern file” > /dev/tty
Elfi test $# -eq 2 ;then
Grep “$1” $2 || echo “$1 not found in $2”>/dev/tty
Else
echo “You didn’t enter two arguments” >/dev/tty
fi
```

It displays the usage when no arguments are input, runs grep if two arguments are entered and displays an error message otherwise.

Run the script four times and redirect the output every time

```
$emp31.sh>foo
Usage : emp.sh pattern file
```

$emp31.sh ftp>foo
You didn't enter two arguments
$emp31.sh henry /etc/passwd>foo
Henry not found in /etc/passwd
$emp31.sh ftp /etc/passwd>foo
ftp:*:325:15:FTP User:/user1/home/ftp:/bin/true

Shorthand for test

[and] can be used instead of test. The following two forms are equivalent

Test $x –eq $y

and

[$x –eq $y]

String Comparison

Test command is also used for testing strings. Test can be used to compare strings with the following set of comparison operators as listed below.

Test	True if
s1=s2	String s1=s2
s1!=s2	String s1 is not equal to s2
-n stg	String stg is not a null string
-z stg	String stg is a null string
stg	String stg is assigned and not null
s1= =s2	String s1=s2

Table: String test used by test

Example:

```
#!/bin/sh
#emp1.sh checks user input for null values finally turns emp.sh developed previously
#

if [ $# -eq 0 ] ; then
echo "Enter the string to be searched :\c"
read pname
if [ -z "$pname" ] ; then
echo "You have not entered th e string"; exit 1
```

```
fi
echo “Enter the filename to be used :\c”
read flname
if [ ! –n “$flname” ] ; then
echo “ You have not entered the flname” ; exit 2
fi
emp.sh “$pname” “$flname”
else
emp.sh $*
fi
```

Output1:

```
$emp1.sh
Enter the string to be searched :[Enter]
You have not entered the string
```

Output2:

```
$emp1.sh
Enter the string to be  searched :root
Enter the filename to be searched :/etc/passwd
Root:x:0:1:Super-user:/:/usr/bin/bash
```

When we run the script with arguments emp1.sh bypasses all the above activities and calls emp.sh to perform all validation checks

```
$emp1.sh jai
You didn’t enter two arguments
```

```
$emp1.sh jai emp,lst
9878|jai sharma|director|sales|12/03/56|70000
```

```
$emp1.sh “jai sharma” emp.lst
You didn’t enter two arguments
```

Because $* treats jai and sharma are separate arguments. And $# makes a wrong argument count. Solution is replace $* with “$@” (with quote” and then run the script.

File Tests

Test can be used to test various file attributes like its type (file, directory or symbolic links) or its permission (read, write. Execute, SUID, etc).

Example:

```
$ ls –l emp.lst
-rw-rw-rw-      1 kumar group          870 jun 8 15:52 emp.lst
$ [-f emp.lst] ; echo $?                              → Ordinary file
0
$ [-x emp.lst] ; echo $?                              → Not an executable.
1
$ [! -w emp.lst]  || echo “False that file not writeable”
False that file is not writable.
```

Example: filetest.sh

```
#! /bin/usr
#
if [! –e $1] : then
        Echo “File doesnot exist”
elif [! –r S1]; then
        Echo “File not readable”
elif[! –w $1]; then
        Echo “File not writable”
else
        Echo “File is both readable and writable”\
fi
```

Output:

$ filetest.sh emp3.lst

File does not exist

$ filetest.sh emp.lst

File is both readable and writable

The following table depicts file-related Tests with test:

Test	True if
-f file	File exists and is a regular file
-r file	File exists and readable
-w file	File exists and is writable
-x file	File exists and is executable
-d file	File exists and is a directory
-s file	File exists and has a size greater than zero
-e file	File exists (Korn & Bash Only)
-u file	File exists and has SUID bit set
-k file	File exists and has sticky bit set
-L file	File exists and is a symbolic link (Korn & Bash Only)
f1 –nt f2	File f1 is newer than f2 (Korn & Bash Only)
f1 –ot f2	File f1 is older than f2 (Korn & Bash Only)
f1 –ef f2	File f1 is linked to f2 (Korn & Bash Only)

Table: file-related Tests with test

The case Conditional

The case statement is the second conditional offered by the shell. It doesn't have a parallel either in C (Switch is similar) or perl. The statement matches an expression for more than one alternative, and uses a compact construct to permit multiway branching. case also handles string tests, but in a more efficient manner than if.

Syntax:

```
case expression in
        Pattern1) commands1 ;;
        Pattern2) commands2 ;;
```

Pattern3) commands3 ;;
…
Esac

Case first matches expression with pattern1. if the match succeeds, then it executes commands1, which may be one or more commands. If the match fails, then pattern2 is matched and so forth. Each command list is terminated with a pair of semicolon and the entire construct is closed with esac (reverse of case).

Example:

```
#! /bin/sh
#
echo “          Menu\n
1. List of files\n2. Processes of user\n3. Today’s Date
4. Users of system\n5.Quit\nEnter your option: \c”
read choice
case “$choice” in
        1) ls –l;;
        2) ps –f ;;
        3) date ;;
        4) who ;;
        5) exit ;;
        *) echo “Invalid option”
esac
```

Output

$ menu.sh

Menu

1. List of files
2. Processes of user
3. Today’s Date
4. Users of system
5. Quit
Enter your option: 3

Mon Oct 8 08:02:45 IST 2007

Note:

- case can not handle relational and file test, but it matches strings with compact code. It is very effective when the string is fetched by command substitution.
- case can also handle numbers but treats them as strings.

Matching Multiple Patterns:

case can also specify the same action for more than one pattern . For instance to test a user response for both y and Y (or n and N).

Example:

```
Echo “Do you wish to continue? [y/n]: \c”
Read ans
Case “$ans” in
        Y | y );;
        N | n) exit ;;
esac
```

Wild-Cards: case uses them:

case has a superb string matching feature that uses wild-cards. It uses the filename matching metacharacters *, ? and character class (to match only strings and not files in the current directory).

Example:

```
Case “$ans” in
        [Yy] [eE]* );;                  Matches YES, yes, Yes, yEs, etc
        [Nn] [oO]) exit ;;              Matches no, NO, No, nO
                *)  echo “Invalid Response”
esac
```

expr: Computation and String Handling

The Broune shell uses expr command to perform computations. This command combines the following two functions:

- Performs arithmetic operations on integers
- Manipulates strings

Computation:

expr can perform the four basic arithmetic operations (+, -, *, /), as well as modulus (%) functions.

Examples:

```
$ x=3 y=5

$ expr  3+5
8

$ expr  $x-$y
-2

$ expr  3 \* 5          Note:\ is used to prevent the shell from interpreting * as metacharacter
15

$ expr  $y/$x
1

$ expr  13%5
3
```

expr is also used with command substitution to assign a variable.

Example1:

```
$ x=6 y=2 : z=`expr $x+$y`
$ echo $z
8
```

Example2:

```
$ x=5
$ x=`expr $x+1`
$ echo $x
6
```

String Handling:

expr is also used to handle strings. For manipulating strings, expr uses two expressions separated by a colon (:). The string to be worked upon is closed on the left of the colon and a regular expression is placed on its right. Depending on the composition of the expression expr can perform the following three functions:

1. Determine the length of the string.
2. Extract the substring.
3. Locate the position of a character in a string.

1. Length of the string:

The regular expression .* is used to print the number of characters matching the pattern .

Example1:

```
$ expr “abcdefg” : ‘.*’
7
```

Example2:

```
while echo “Enter your name: \c” ;do
        read name
        if [‘expe “$name” :’.*’‘    -gt  20] ; then
                echo “Name is very long”
        else
                break
        fi
done
```

2. Extracting a substring:

expr can extract a string enclosed by the escape characters \ (and \).

Example:

```
$ st=2007
$ expr “$st” :’..\(..\)’
07                                    Extracts last two characters.
```

3. **Locating position of a character:**

   expr can return the location of the first occurrence of a character inside a string.

   Example:

   $ stg = abcdefgh ; expr “$stg” : ‘[^d]*d’
   4 Extracts the p*osition of character d*

$0: Calling a Script by Different Names

There are a number of UNIX commands that can be used to call a file by different names and doing different things depending on the name by which it is called. $0 can also be to call a script by different names.

Example:

```
#! /bin/sh
#
lastfile=`ls –t *.c |head -1`
command=$0
exe=‘expr $lastfile: ‘\(.*\).c’‘
case $command in
        *runc) $exe ;;
        *vic) vi $lastfile;;
        *comc) cc –o $exe $lastfile &&
                Echo “$lastfile compiled successfully”;;
esac
```

After this create the following three links:

```
ln comc.sh comc
ln comc.sh runc
ln comc.sh vic
```

Output:

```
$ comc
hello.c compiled successfully.
```

While: Looping

To carry out a set of instruction repeatedly shell offers three features namely while, until and for.

Synatx:

```
while condition is true
do
        Commands
done
```

The commands enclosed by do and done are executed repadetedly as long as condition is true.

Example:

```
  #! /bin/usr
 ans=y
while [“$ans”=”y”]
do
        echo “Enter the code and description : \c” > /dev/tty
        read code description
        echo “$code $description” >>newlist
        echo “Enter any more [Y/N]”
        read any
        case $any in
        Y* | y* ) answer =y;;
        N* | n*) answer = n;;
                *) answer=y;;
        esac
done
```

Input:

Enter the code and description : 03 analgestics

Enter any more [Y/N] :y

Enter the code and description : 04 antibiotics

Enter any more [Y/N] : [Enter]

Enter the code and description : 05 OTC drugs

Enter any more [Y/N] : n

Output:

$ cat newlist

03 | analgestics

04 | antibiotics

05 | OTC drugs

Other Examples: An infinite/semi-infinite loop

(1)

```
while true ; do
   [ -r $1 ] && break
   sleep $2
done
```

(2)

```
while [ ! -r $1 ] ; do
   sleep $2
done
```

for: Looping with a List

for is also a repetitive structure.

Synatx:

for variable in list

do

Commands

done

list here comprises a series of character strings. Each string is assigned to variable specified.

Example:

```
for file in ch1 ch2; do
> cp $file ${file}.bak
> echo $file copied to $file.bak
done
```

Output:

```
ch1 copied to ch1.bak
ch2 copied to ch2.bak
```

Sources of list:

- **List from variables**: Series of variables are evaluated by the shell before executing the loop

  Example:

  $ for var in $PATH $HOME; do echo "$var" ; done

  Output:
  /bin:/usr/bin;/home/local/bin;
  /home/user1

- **List from command substitution**: Command substitution is used for creating a list. This is used when list is large.

  Example:

  $ for var in `cat clist`

- **List from wildcards**: Here the shell interprets the wildcards as filenames.

  Example:

```
for file in *.htm *.html ; do
        sed 's/strong/STRONG/g
        s/img src/IMG SRC/g' $file > $$
        mv $$ $file
done
```

- **List from positional parameters**:

  Example: emp.sh

  ```
  #! /bin/sh
  for pattern in "$@"; do
  grep "$pattern" emp.lst || echo "Pattern $pattern not found"
  done
  ```

  Output:

  $emp.sh 9876 "Rohit"

9876	Jai Sharma	Director	Productions
2356	Rohit	Director	Sales

basename: Changing Filename Extensions

They are useful in chaining the extension of group of files. Basename extracts the base filename from an absolute pathname.

Example1:

$basename /home/user1/test.pl

Ouput:

test.pl

Example2:

$basename test2.doc doc

Ouput:

test2

Example3: Renaming filename extension from .txt to .doc

```
for file in *.txt ; do
     leftname=`basename $file .txt` Stores left part of filename
      mv $file ${leftname}.doc
done
```

set and shift: Manipulating the Positional Parameters

The set statement assigns positional parameters $1, $2 and so on, to its arguments. This is used for picking up individual fields from the output of a program.

Example 1:

$ set 9876 2345 6213

$

This assigns the value 9876 to the positional parameters $1, 2345 to $2 and 6213 to $3. It also sets the other parameters $# and $*.

Example 2:

$ set `date`

$ echo $*

Mon Oct 8 08:02:45 IST 2007

Example 3:

$ echo “The date today is $2 $3, $6”

The date today is Oct 8, 2007

Shift: Shifting Arguments Left

Shift transfers the contents of positional parameters to its immediate lower numbered one. This is done as many times as the statement is called. When called once, $2 becomes $1, $3 becomes S2 and so on.

Example 1:

$ echo “$@” *$@ and $* are interchangeable*

Mon Oct 8 08:02:45 IST 2007

$ echo $1 $2 $3

Mon Oct 8

$shift

$echo $1 $2 $3

Mon Oct 8 08:02:45

$shift 2 *Shifts 2 places*

$echo $1 $2 $3

08:02:45 IST 2007

Example 2: emp.sh

```
#! /bin/sh
Case $#  in
        0|1) echo “Usage: $0 file pattern(S)” ;exit ;;
          *) fname=$1
           shift
           for pattern in “$@” ; do
                  grep “$pattern” $fname || echo “Pattern $pattern not found”
        done;;
esac
```

Output:

$emp.sh emp.lst

Insufficient number of arguments

$emp.sh emp.lst Rakesh 1006 9877

9876 Jai Sharma Director Productions

2356 Rohit Director Sales

Pattern 9877 not found.

Set -- : Helps Command Substitution

Inorder for the set to interpret - and null output produced by UNIX commands the – option is used . If not used – in the output is treated as an option and set will interpret it wrongly. In case of null, all variables are displayed instead of null.

Example:

$set `ls –l chp1`

Output:

-rwxr-xr-x: bad options

Example2:

$set `grep usr1 /etc/passwd`

Correction to be made to get correct output are:

$set -- `ls –l chp1`

$set -- `grep usr1 /etc/passwd`

The Here Document (<<)

The shell uses the << symbol to read data from the same file containing the script. This is referred to as a here document, signifying that the data is here rather than in aspirate file. Any command using standard input can slo take input from a here document.

Example:

```
mailx kumar << MARK
Your program for printing the invoices has been executed
on `date`.Check the print queue
The updated file is $flname
MARK
```

The string (MARK) is delimiter. The shell treats every line following the command and delimited by MARK as input to the command. Kumar at the other end will see three lines of message text with the date inserted by command. The word MARK itself doesn't show up.

Using Here Document with Interactive Programs:

A shell script can be made to work non-interactively by supplying inputs through here document.

Example:

```
$ search.sh << END
> director
>emp.lst
>END
```

Output:

Enter the pattern to be searched: Enter the file to be used: Searching for director from file emp.lst

9876	Jai Sharma	Director	Productions
2356	Rohit	Director	Sales

Selected records shown above.

The script search.sh will run non-interactively and display the lines containing “director” in the file emp.lst.

trap: interrupting a Program

Normally, the shell scripts terminate whenever the interrupt key is pressed. It is not a good programming practice because a lot of temporary files will be stored on disk. The trap statement lets you do the things you want to do when a script receives a signal. The trap statement is normally placed at the beginning of the shell script and uses two lists:

trap ‘command_list’ signal_list

When a script is sent any of the signals in signal_list, trap executes the commands in command_list. The signal list can contain the integer values or names (without SIG prefix) of one or more signals – the ones used with the kill command.

Example: To remove all temporary files named after the PID number of the shell:

trap ‘rm $$* ; echo “Program Interrupted” ; exit’ HUP INT TERM

trap is a signal handler. It first removes all files expanded from $$*, echoes a message and finally terminates the script when signals SIGHUP (1), SIGINT (2) or SIGTERM(15) are sent to the shell process running the script.

A script can also be made to ignore the signals by using a null command list.

Example:

trap ‘’ 1 2 15

Programs

1)

```
#!/bin/sh
IFS="|"
While echo "enter dept code:\c"; do
Read dcode
Set -- ‘grep "^$dcode"<<limit
01|ISE|22
02|CSE|45
03|ECE|25
04|TCE|58
limit`
Case $# in
3) echo "dept name :$2 \n  emp-id:$3\n"
*) echo "invalid code";continue
esac
done
```

Output:

```
$valcode.sh
Enter dept code:88
Invalid code
Enter dept code:02
Dept name  : CSE
Emp-id :45
Enter dept code:<ctrl-c>
```

2)
```
#!/bin/sh
x=1
While [$x –le 10];do
 echo “$x”
 x=`expr $x+1`
done
#!/bin/sh
sum=0
for I in “$@” do
 echo “$I”
sum=`expr $sum + $I`
done
Echo “sum is $sum”
```

3)
```
#!/bin/sh
sum=0
for I in `cat list`; do
 echo “string is $I”
x= ‘expr “$I”:’.*’‘
Echo “length is $x”
Done
```

4)
This is a non-recursive shell script that accepts any number of arguments and prints them in a reverse order.

For example if A B C are entered then output is C B A.

```
#!/bin/sh
if [ $# -lt 2 ]; then
echo "please enter 2 or more arguments"
exit
fi
for x in $@
do
y=$x" "$y
done
echo "$y"
```

Run1:
```
[root@localhost shellprgms]# sh sh1a.sh 1 2 3 4 5 6 7

7 6 5 4 3 2 1
```

Run2: [root@localhost shellprgms]# sh ps1a.sh this is an argument

argument an is this

5)

The following shell script to accept 2 file names checks if the permission for these files are identical and if they are not identical outputs each filename followed by permission.

```
#!/bin/sh
if [ $# -lt 2 ]
then
echo "invalid number of arguments"
exit
fi
str1=`ls -l $1|cut -c 2-10`
str2=`ls -l $2|cut -c 2-10`

if [ "$str1" = "$str2" ]
then
echo "the file permissions are the same: $str1"
else
echo " Different file permissions "
echo -e "file permission for $1 is $str1\nfile permission for $2 is  $str2"
fi
```

Run1:

```
[root@localhost shellprgms]# sh 2a.sh ab.c xy.c
file permission for ab.c is rw-r--r--
file permission for xy.c is  rwxr-xr-x
```

Run2:

[root@localhost shellprgms]# chmod +x ab.c

[root@localhost shellprgms]# sh 2a.sh ab.c xy.c

the file permissions are the same: rwxr-xr-x

6)This shell function that takes a valid directory name as an argument and recursively descends all the subdirectories, finds the maximum length of any file in that hierarchy and writes this maximum value to the standard output.

```
#!/bin/sh
if [ $# -gt 2 ]
then
echo "usage sh flname dir"
exit
fi
if [ -d $1 ]
then
ls -lR $1|grep -v ^d|cut -c 34-43,56-69|sort -n|tail -1>fn1
echo "file name is `cut -c 10- fn1`"
echo " the size is `cut -c -9 fn1`"
else
echo "invalid dir name"
fi
```

Run1:

[root@localhost shellprgms]# sh 3a.sh

file name is a.out

the size is 12172

7)This shell script that accepts valid log-in names as arguments and prints their corresponding home directories. If no arguments are specified, print a suitable error message.

```
if [ $# -lt 1 ]
then
echo " Invlaid Arguments..... "
exit
fi
for x in "$@"
```

```
do
  grep -w "^$x" /etc/passwd | cut -d ":" -f 1,6
done
```

Run1:

```
[root@localhost shellprgms]# sh 4a.sh root
root:/root
```

Run2:

```
[root@localhost shellprgms]# sh 4a.sh
Invalid Arguments.....
```

8) This shell script finds and displays all the links of a file specified as the first argument to the script. The second argument, which is optional, can be used to specify the directory in which the search is to begin. If this second argument is not present .the search is to begin in current working directory.

```
#!/bin/bash
if [ $# -eq 0 ]
then
        echo "Usage:sh 8a.sh[file1] [dir1(optional)]"
        exit
fi
if [ -f $1 ]
then
        dir="."
if [ $# -eq 2 ]
then
        dir=$2
fi
inode=`ls -i $1|cut -d " " -f 2`
echo "Hard links of $1 are"
find $dir -inum $inode -print
```

```
echo "Soft links of $1 are"
find $dir -lname $1 -print
else
echo "The file $1 does not exist"
fi
```

Run1:

```
[root@localhost shellprgms]$ sh 5a.sh  hai.c
Hard links of hai.c are
./hai.c
Soft links of hai.c are
./hai_soft
```

9) This shell script displays the calendar for current month with current date replaced by * or ** depending on whether date has one digit or two digits.

```
#!/bin/bash
n=` date +%d`
echo " Today's date is : `date +%d%h%y` ";
cal > calfile
if [ $n -gt 9 ]
then
 sed "s/$n/\**/g" calfile
else
 sed "s/$n/\*/g" calfile
[root@localhost shellprgms]# sh 6a.sh
 Today's date is : 10 May 05
      May 2005
Su  Mo  Tu  We  Th  Fr  Sa
 1   2    3   4   5   6   7
 8   9   **  11  12  13  14
15  16   17  18  19  20  21
```

22 23 24 25 26 27 28

29 30 31

10) This shell script implements terminal locking. Prompt the user for a password after accepting, prompt for confirmation, if match occurs it must lock and ask for password, if it matches terminal must be unlocked

```
trap “ " 1 2 3 5 20
clear
echo -e “\nenter password to lock terminal:"
stty -echo
read keynew
stty echo
echo -e “\nconfirm password:"
stty -echo
read keyold
stty echo
if [ $keyold = $keynew ]
then
echo "terminal locked!"
while [ 1 ]
do
echo "retype the password to unlock:"
stty -echo
read key
if [ $key = $keynew ]
then
stty echo
echo "terminal unlocked!"
stty sane
exit
fi
echo "invalid password!"
done
else
```

```
echo " passwords do not match!"
fi
stty sane
```

Run1:

```
[root@localhost shellprgms]# sh 13.sh
enter password:
confirm password:
terminal locked!
retype the password to unlock:
invalid password!
retype the password to unlock:
terminal unlocked!
```

*****